Sujet systèmes embarqués – examen M2

J.-M Friedt, 10 février 2016

Tous les documents sont autorisés, toutes les connexions à internet sont autorisées mais les communications par téléphone mobile sont **proscrites**.

Les nombres entre parenthèses à la fin de chaque question indiquent le nombre de points qu'une bonne réponse fournira. À l'issue de l'examen, un document papier résumera les principales étapes nécessaires pour compléter chaque question, en plus duquel il est souhaitable d'envoyer une archive organisant *clairement* les programmes répondant aux attentes de chaque partie (un sous-répertoire par question peut fournir une solution).

Le sujet porte sur la création de modules noyau Linux, leur portabilité, et leur utilisation pour l'accès cohérent aux ressources matérielles par les mécanismes de protection fournies par les diverses API des pilotes Linux.

1 `/dev`

Historiquement, selon la philosophie que tout accès aux ressources sous Unix doit apparaître comme un accès à un fichier, un point d'entrée spécifique sous `/dev` contient les pseudo-fichiers chargés de communiquer avec le noyau et ses pilotes (modules).

1. Rappeler comment les divers point d'accès de `/dev` sont identifiés par le noyau (1).
2. En particulier, quelle est la valeur de cet identifiant pour un port série accessible par `/dev/ttyS*` ? (1)
3. Quelles sont les fonctions à implémenter dans un module noyau pour permettre l'interaction avec l'espace utilisateur ? (1)
4. Comment identifier la version du noyau en cours d'exécution sur une plateforme exécutant GNU/Linux ? Quelle réponse obtenons-nous sur le PC ? sur la carte A13 ? En déduire l'emplacement de l'arborescence des sources du noyau sur le PC. (1)
5. Démontrer l'écriture et la compilation d'un module noyau excessivement simple qui ne fait rien d'autre qu'afficher dans le log du système `Bonjour` lors de son chargement et `Adieu` lors de son retrait de la mémoire. (2)
6. Comment afficher (commande shell en espace utilisateur) les messages du système permettant de valider le bon fonctionnement du module ? (1)
7. Ajouter au module la capacité à répondre par "Hello" à une requête de lecture. Démontrer son fonctionnement avec une commande exécutée depuis le shell et sollicitant en lecture le point d'entrée dans le répertoire `/dev/` adéquat. (2)
8. Proposer un programme en C (en espace utilisateur) réalisant la même fonction de lecture du résultat depuis le point d'entrée dans `/dev`. (2)

2 `/sys/class` et le *devicetree*

Plus récemment, les développeurs du noyau Linux proposent une interface permettant une communication plus homogène avec les périphériques – les fichiers permettant la configuration d'une classe donnée de périphériques sont standardisés et ne nécessitent plus de retrouver les arguments aux `ioctl` de chaque pilote. Par ailleurs, ces nouvelles interfaces permettent une communication en trames ASCII – compatible avec les langages scriptés tels que le shell ou Python – au lieu de transactions binaires.

Après être passé du script de configuration du matériel `script.fex` que nous avons vu en cours vers le fichier de configuration `devicetree.dts`, le mode d'accès vers les ports n'est plus d'exporter un numéro de GPIO indiqué dans `script.fex` mais d'exporter un numéro de broche égal à $N \times 32 + P$ avec N l'indice du port ($A = 0$, $B = 1$, ...) et P le numéro de la broche.

9. Tenter d'accéder au port PG9 qui supporte la LED commandable depuis un GPIO. Que constatez-vous ? (1)
10. En cas d'échec de la question précédente (justifier), tenter d'accéder à la broche 14 du connecteur GPIO2 (pin 35 de la carte) de la carte Olinuxino-A13micro, et démontrer l'allumage et l'extinction d'une LED qui y est connectée depuis l'espace utilisateur. (2)

3 IIO

Nous avons vu que Industrial Input/Output (IIO) fournit une infrastructure portable, visant à homogénéiser les communications avec les interfaces de même type (accéléromètre, convertisseur numérique-analogique ou analogique-numérique ...) : son implémentation se trouve en particulier dans `drivers/iio/` des sources du noyau Linux.

Un pilote compatible IIO est disponible à `http://jmfriedt.free.fr/M2_iio.tar.gz`

11. Compiler le pilote sur PC et démontrer son chargement en mémoire en décrivant le message affiché dans les logs du système. (1)
12. Idem sur plateforme A13. On notera que la version de `buildroot` qui a servi à compiler l'image active sur la carte A13 mise à disposition se trouve dans `/home/jmfriedt/buildroot`. (1)

Le point le plus complexe de la lecture d'un pilote supportant les plateformes ou la structure IIO est l'imbrication des macros, qui d'un côté rend le code plus compact, mais d'un autre côté rend l'identification des erreurs complexes si une macro n'est pas convenablement déployée. Par exemple, dans la macro `#define COMPOSANT_CHANNEL(_chan, _bits) { \` qui prend deux arguments (l'indice de la voie et sa résolution), les attributs
`.info_mask_separate = BIT(IIO_CHAN_INFO_RAW), \` sont individuels à chaque voie ainsi définie (ici la valeur du canal), tandis que les attributs
`.info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE), \` seront communs à toute les voies (ici le coefficient reliant bits et tension par exemple). Par ailleurs, comme la `fops` d'un pilot communiquant par `/dev`, une structue définit les fonctions appelées par les diverses méthodes susceptibles d'accéder aux pseudo-fichiers : ainsi, ici `.read_raw` de `static const struct iio_info` pointe vers la fonction `composant_read_raw()`. Cette fonction place dans `*val` le résultat fourni par l'utilisateur qui sollicite le canal en lecture, tandis que la valeur retournée par la fonction indique de la nature de la transaction (`IIO_VAL_INT` si la valeur retournée doit être considérée comme un entier, `IIO_VAL_FRACTIONAL_LOG2` s'il s'agit d'un flottant dont une partie se trouve dans `val2`)). De même, l'argument `val` est fourni depuis l'espace utilisateur lorsque le pseudo-fichier est sollicité en écriture en se traduisant par l'appel à la fonction `composant_write_raw()`. Ces diverses considérations se retrouvent dans les sources du pilote `iio` et en particulier dans la fonction `iio_format_value()` de `drivers/iio/industrialio-core.c` dans les sources du noyau.

13. Rappeler le rôle de `dummy_platform.c` – en particulier de la fonction `platform_device_register()` – et sa relation à `composant.c` (1)
14. Combien de voies de communication ce pilote supporte-t-il ? Quelle valeur renvoie une lecture sur chaque voie ? (1)
15. Ajouter le support pour une voie additionnelle, qui renvoie lors de la lecture du nouveau point d'entrée (au format `voltage_raw`), son numéro de canal. (2)
16. Modifier le pilote afin que lors d'une écriture dans ce pseudo-fichier, la valeur écrite dans un premier temps (`echo "valeur" > fichier`) soit mémorisée, puis restituée dans un second temps lors d'une lecture. (2)

Correction systèmes embarqués – examen M2

J.-M Friedt, 10 février 2016

Le sujet porte sur la création de modules noyau Linux, leur portabilité, et leur utilisation pour l'accès cohérent aux ressources matérielles par les mécanismes de protection fournies par les diverses API des pilotes Linux.

4 /dev

1. Chaque classe d'entrée dans le répertoire `/dev` est identifiée par son identifiant majeur (*major number*) et le nœud au sein de chaque classe par son identifiant mineur (*minor number*) qui s'incrémente pour chaque élément de la classe.
2. ```
   crw-rw---T 1 root dialout 4, 64 Dec 18 06:37 /dev/ttyS0
   crw-rw---T 1 root dialout 4, 65 Dec 18 06:37 /dev/ttyS1
   crw-rw---T 1 root dialout 4, 66 Dec 18 06:37 /dev/ttyS2
   crw-rw---T 1 root dialout 4, 67 Dec 18 06:37 /dev/ttyS3
   ```
   Le major number des périphériques liés aux ports série est 4.
3. Il faudra `read` et `write` pour communiquer, ainsi que `open` et `release` pour accéder au fichier.
4. `uname` avec l'option `-v` (kernel version) ou `-a` (all). Il faut par conséquent modifier le Makefile pour adapter l'option `-C` et le faire pointer vers `/usr/src/linux-headers-3.16.0-4-686-pae` (à la date de rédaction de ce document).
5. ```
   #include <linux/module.h>
   #include <linux/kernel.h>
   #include <linux/init.h>
   static int __init hello_start(void){printk(KERN_INFO "Bonjour");return 0;}
   static void __exit hello_end(void) {printk(KERN_INFO "Adieu");}
   module_init(hello_start);
   module_exit(hello_end);
   ```
6. `dmesg`
7. ```
   // mknod /dev/jmf c 90 0
   #include <linux/module.h>       /* Needed by all modules */
   #include <linux/kernel.h>       /* Needed for KERN_INFO */
   #include <linux/init.h>         /* Needed for the macros */
   #include <linux/fs.h>           // define fops
   #include <asm/uaccess.h>

   static int dev_open(struct inode *inod,struct file *fil);
   static int dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
   static int dev_rls(struct inode *inod,struct file *fil);

   int hello_start(void); // declaration pour eviter les warnings
   void hello_end(void);

   static struct file_operations fops=
   {.read=dev_read,
    .open=dev_open,
    .release=dev_rls,
   };

   int hello_start()  // init_module(void)
   {int t=register_chrdev(90,"jmf",&fops);  // major = 90
    printk(KERN_INFO "Bonjour\n");
    return t;
   }

   void hello_end() // cleanup_module(void)
   {printk(KERN_INFO "Adieu\n");
    unregister_chrdev(90,"jmf");
   }

   static int dev_rls(struct inode *inod,struct file *fil) {return 0;}
   static int dev_open(struct inode *inod,struct file *fil) {return 0;}

   static int dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
   {char buf[15]="Hello\n\0";
    int readPos=0;
    printk(KERN_ALERT "read\n");
    while (len && (buf[readPos]!=0)) {put_user(buf[readPos],buff++);readPos++;len--;}
    return readPos;
   }
   ```
   ```

```
module_init(hello_start);
module_exit(hello_end);
```

8. `int main() {char t[42];f=fopen("/dev/jmf","r");d=fscanf(f,"%s",t);printf("%s,t");}`

5 /sys/class

1. Une tentative d'accès à la broche d'indice 201 ($6 \times 32 + 9$) se traduit par un échec car le pilote gérant la LED occupe déjà la ressource (`/sys/class/led/*`).
2. La broche 12 du connecteur GPIO2 est liée à PE10 qui est donc indexée par 138.

6 IIO

On notera qu'il est possible de générer le code issu du préprocesseur, lors de la compilation du module noyau, en ajoutant l'option `ccflags-y:=-E` dans le Makefile. Bien que cette option induise un échec de la compilation, le fichier d'extension `.o` généré contient le résultat issu du préprocesseur, incluant les macros qui ont été interprétées. Le résultat est illisible !

1. Compiler sur PC nécessite de renseigner l'emplacement des sources du noyau exécuté sur le PC, tel que nous l'avons vu auparavant.
   Par ailleurs, IIO n'est pas supporté par défaut par le noyau, tels que l'indiquent les messages

   ```
   [639540.497546] composant: Unknown symbol devm_iio_device_alloc (err 0)
   [639540.497574] composant: Unknown symbol iio_device_unregister (err 0)
   [639540.497596] composant: Unknown symbol iio_device_register (err 0)
   ```

   Il faut donc charger le module par `insmod industrialio`.
   Insérer `dummy_platform.ko` se traduit par

   ```
   [10224.335218] board mounted
   ```

   tandis que insérer `composant.ko` se traduit par

   ```
   [10165.062818] . Entering probe
   [10165.062846] . Registering
   [10165.062911] . Registered
   ```

2. Ici encore il faut prendre soin de faire pointer le Makefile vers les sources du noyau, qui se trouvent dans
   `$HOME/buildroot/output/build/linux-fbe157ba74f80634ece3b8e55d28ded467ffe821/`

1. en chargeant le module, nous constatons dans `/sys/bus/iio/devices/iio\:device0/` la présence de `out_voltage0_raw` et `out_voltage1_raw` : le pilote supporte un composant à deux voies de communication. Ces deux voies renvoient, à la lecture, leur indice :

   ```
   # cat /sys/bus/iio/devices/iio\:device0/out_voltage*raw
   0
   1
   ```

2. Une voie additionnelle s'ajoute en complétant dans le fichier d'entête `#define COMPOSANT_CHANNELS 2` et

   ```
   #define DECLARE_COMPOSANT_CHANNELS(_name, _bits) \
           const struct iio_chan_spec _name##_channels[] = { \
                   COMPOSANT_CHANNEL(0, _bits), \
                   COMPOSANT_CHANNEL(1, _bits), \
                   COMPOSANT_CHANNEL(2, _bits), \
   }
   ```

   Nous le constatons par le passage de

   ```
   # ls /sys/bus/iio/devices/iio\:device0/
   dev  name  out_voltage0_raw  out_voltage1_raw  power  subsystem  uevent
   ```

   à

   ```
   # ls /sys/bus/iio/devices/iio\:device0/
   dev  name  out_voltage0_raw  out_voltage1_raw  out_voltage2_raw  power  subsystem  uevent
   ```

3. le code

```
int mesure=0;

static int composant_read_raw(struct iio_dev *indio_dev,
                           struct iio_chan_spec const *chan,
                           int *val, int *val2, long m)
{
        struct composant_state *st = iio_priv(indio_dev);

        switch (m) {
        case IIO_CHAN_INFO_RAW:
                *val=mesure;
                return IIO_VAL_INT;
        }
        return -EINVAL;
}

static int composant_write_raw(struct iio_dev *indio_dev,
                               struct iio_chan_spec const *chan,
                               int val, int val2, long mask)
{ printk(KERN_ALERT "Write\n");
  switch (mask) {
  case IIO_CHAN_INFO_RAW:
       printk(KERN_ALERT "Write raw\n");
       mesure=val;
  }
  return 0;
}
```

se traduit par

```
# echo "42" > /sys/bus/iio/devices/iio\:device0/out_voltage0_raw
# cat  /sys/bus/iio/devices/iio\:device0/out_voltage0_raw
42
# echo "99" > /sys/bus/iio/devices/iio\:device0/out_voltage0_raw
# cat  /sys/bus/iio/devices/iio\:device0/out_voltage0_raw
99
```